Reference urls

|  |  |
| --- | --- |
| bharath sir projects | <https://github.com/manideep-vv/forked-javawebservices-bharath>  but this projects were with spr boot 2.5 only & I have downloaded latest projects |
| cxf docs | <https://cxf.apache.org/> tells which is latest cxf version  <https://cxf.apache.org/docs/> for all hyper links  <https://cxf.apache.org/docs/a-simple-jax-ws-service.html>  <https://cxf.apache.org/docs/springboot.html>  <https://cxf.apache.org/docs/index.html>  <https://docs.redhat.com/en/documentation/red_hat_fuse/7.5/html-single/apache_cxf_development_guide/index>  <https://cxf.apache.org/resources-and-articles.html> |
|  | <https://www.blackslate.io/articles/getting-started-with-apache-cxf-for-soap-web-services-in-java> |
| to see existing issues in cxf | <https://issues.apache.org/jira/projects/CXF/issues/CXF-9116?filter=allopenissues> |
| security | <https://cxf.apache.org/docs/ws-security.html> |
| json official doc | <https://www.json.org/json-en.html> |
| books | java and soap – for debugging |

Current status-

unable to start app with switched to 21- but check if app is running or not

complete section 4 & 5 – to know about XSD file

solve that user name token program use a runner

SOAP webervices & advantages

JAX-WS – java api for xml web services

JAX-RS java api for restful web services

|  |  |
| --- | --- |
| SOAP= JAX-WS  REST = JAX-RS  JAX-WS – java api for xml using webservices  SOAP - simple object access protocol  JAX-WS - java api for xml web services  JAX-RS - java api for restful services  Generally, specification means set of rules, guidelines (english statements), API is set of interfaces  JAX-WS | Heterogenenous apps/ Cross platform apps can communicate with each other (java app can talk to .net app) as below   * This communication is possible only bec data exchange is happ in xml and WSDL (WSDL is an xml file with which stubs are being generated )   java SOAP producer will develop the project, he will give us XML WSDL file to clients (either java/ .net/ any client)  those clients will generate the stubs (java clients will generate java stubs & .net guys will .net classes)  and those clients/consumers can just call those classes  since those stubs have the url already- if u call the stubs then real service will be called  so with this approach java app (soap consumer) can talk to .net soap producer |

A soap web service is a implementation of independent business functionality

In SOA service oriented architecture, data will transmitted with xml only

Webservices is one of the way in which SOA can be implemented

standards

|  |  |
| --- | --- |
| WS-security |  |
| MTOM- Message Transmission and Optimization Mechanism. | is to exchange files – like any file – images, pdf/excel file  <https://cxf.apache.org/docs/mtom.html> |
| WS- adressing – | for async callbacks & to redirect the response to a diff server instead of the one who called it |
| WS-policy- | assert and mandate certain rules to consume web services  ex:- all consumers should use HTTPS to call a webservice and this module ensures if any consumer still uses http then they will throw exceptions |

These frameworks are responsible for serializing and de-serialinzg xml/ JSON into java objects internally

SOAP only uses XML for processing

Designing approaches

|  |  |
| --- | --- |
| Contract first / WSDL first /Top Down design (complex approach as we need to write WSDL file on our own) | Code first /Bottom up |
| * in this 1st we will design the contract /WSDL, * Then generate java stubs, and write implementation classes for the generated stub interfaces and client * can provide sign off for WSDL file * to generate stubs/java classes from wsdl file, use plugin “cxf-codegen-plugin”   <groupId>org.apache.cxf</groupId>  <artifactId>cxf-codegen-plugin</artifactId>  <version>3.3.0</version>  ex:- this plugin will generate even soap controller Interface, developers needs to write an impl class for that generated SOAPController/SOAPWebservice class  my view- for RESTFul web services also   * using open api, first we can write the open API spec, * then we can generate stubs from openAPI spec yam file, * Here also clients can provide sign off for openAPI spec YML file | 1st write code  then generate WSDL file using “java2wsdl” tool  mostly preffered for legacy code base |

General advantages of webservices

(SOAP / RESTful web services)

-----------------------------

1. with webservices app - 1 app can talk to another apps, like java app can talk to a webservice written in PHP

when to use SOAP web services?

------------------------

1) when we want a formal contract like WSDL-

WSDL only for SOAP,

my view - But for RESTful webservices- we have Open API swagger and WADL(this is less commonly used than Open API)

2) security, Transaction management - these 2 will come out of the box

reliable async process out of the box

my view- spring also can provide security and wonderful transaction management

3) Reliable messaging - SOAP supports WS-ReliableMessaging, ensuring that messages are delivered reliably and in the correct order.

This is crucial for applications where data integrity is paramount

my view - kafka also can send the messages async/ spring web client also can make async calls reliably if we use resilience 4j we can even do re-try mechanism also

problem with soap webservices

---------

1. since web services uses only XML- all data is transferred in XML, serialization & xml parsing is a performance over head
2. and xml is case sensitive, even if u change small case in xml , meaning will change

<mani> and <Mani> these 2 tags are different

<name> sai </Name> -- now this will give error saying , tag is not closed properly

Adv is – cross plat communication which is possible in RESTful web services also

Most important interesting things

1. with single annotation you can log both the request and response into console- which is lovely things

@Features(features = "org.apache.cxf.feature.LoggingFeature")  
public class StudentSOAPController {

1. WSDL file will be automatically generated, and we can access them from url “http://localhost:8080/services”

|  |  |
| --- | --- |
|  |  |

Analogy- at every temple entrance we should keep 1 male and female security guard and those people are specific to that door only not for all doors

1. For every endpoint- we can configure both interceptor and handler

Both interceptor and handlers will intercept and process the incoming message

And these configuration is separate for each endpoint

Ex:- if u attach a interceptor and handler for 1 endpoint It means those are exclusive for that endpoint, and these handlers will not be fired for other endpoint

XML & XSD & JSON

In html <h1> Manideep</h2> most of the tags are predefined and these tags are having a special meaning and provide some additional func to the data

Where as in xml we don’t have any pre-defined tags

XML

Markup in XML refers to the extra characters(start and ending tags) and structures within an XML document that provide information about the data. Think of it as the "instructions" or "metadata" that describe the content. It's what differentiates the actual data from the structural elements of the XML document

Here are the main types of markup in XML:

* **Start-tags:** These mark the beginning of an XML element. They are enclosed in angle brackets and include the element's name (e.g., <book>).
* **End-tags:** These mark the end of an XML element. They are similar to start-tags but include a forward slash before the element name (e.g., </book>). Every start-tag must have a corresponding end-tag for the XML to be well-formed.

XSD

In xml namespace is nothing but package name

An XML can have only elements from a multiple namespaces

Instead of using the entire namespace url to refer to a xml element we can use a prefix

|  |  |
| --- | --- |
| Xsd means xml schema definition, XSD will have following  what elements names are allowed, what should be the type of each ele (int/string..)  what attributes are allowed for a xml tag,  sequence/order of xml elements,  num of occurances of each ele  namespaces | if any xml is following the xsd syntax then it is called valid xml    here when we are entering data in xml, it will always be validated against xsd and tell us, whether xml data is correct or not |
| sample xsd file content    <complexType name="Patient">  <sequence>  <element name="name" type="tns:String15Chars"/>  <element name="age" type="int"/>  <element name="dob" type="date"/>  <element name="email" type="string" maxOccurs="unbounded"/>  <element name="gender" type="tns:Gender"/>  <element name="phone" type="string"/>  <element name="payment" type="tns:PaymentType"/>  </sequence>  <attribute name="id" type="tns:ID"/>  </complexType>  <simpleType name="String15Chars">  <restriction base="string">  <maxLength value="15" />  </restriction>  </simpleType> | see here we have defined the type of each ele in xsd  examples of xsd validation  <element name="email" type="string" maxOccurs="unbounded" />  here maxoccurs unbounded means this ele can be present any number of times in xml  <element name="email" type="string" minOccurs="0" /> here 0 means this ele is optional, even if this ele is optional compiler wont complain about absence of this ele  what is elementFormDefault in xsd    this qualified value attribute tells the parser to check for namespace in xml content    see here for every ele, “tns” namespace/package name is mandatory as we gave qualified in xsd  how an xml is validated against an XSD?  <tns:patient  xmlns:tns="http://bharaththippireddy.com/Patient"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://bharaththippireddy.com/Patient Patient.xsd " id="123">  so here in xml we will give the xsd file location, using that it will validate |
| sub element vs type  <payment type="cash" > //here this type is called attributes of a tag  <insurance> //these are called sub elements of root element  <providerName>Blue Cross</providerName>  <limit>1000</limit>  </insurance>  </payment> |  |

JSON is a very light weight data format which is used to exchange the data between apps

JAXB Marshalling and un-marshalling

JAXB is the specification defined by oracle (marshalling means process of converting java obj data to xml payload )

XJB= means xml to java binding

Java architecture for xml binding, this jaxb can transform the xml payload to java objects (like how hibernate maps java objects to db rows)

Xml data will be used to communicate between SOAP web services

Top of Form

JAXB is easy to use than directly working with XML through DOM or SAX parsing

Bottom of Form

|  |  |  |
| --- | --- | --- |
|  |  |  |

Xml schema def

|  |  |
| --- | --- |
| **Below is the xml schema definition – which contains field names & field type**  **<!-- XML Schema mapping for USAddress -->**  <xs:complexType name="USAddress">  <xs:sequence>  <xs:element name="street" type="xs:string"/>  <xs:element name="city" type="xs:string"/>  <xs:element name="state" type="xs:string"/>  <xs:element name="zip" type="xs:decimal"/>  <xs:element name="name" type="xs:string"/>  </xs:all>  </xs:complexType> | @XmlType(propOrder={"street", "city" , "state", "zip", "name" })  public class USAddress {  String getName() {..};  void setName(String) {..};  String getStreet() {..};  void setStreet(String) {..};  String getCity() {..};  void setCity(String) {..};  String getState() {..};  void setState(String) {..};  java.math.BigDecimal getZip() {..};  void setZip(java.math.BigDecimal) {..};  } |

3 tools given by jaxb api

|  |  |
| --- | --- |
| XJC- this xjc tool is to convert xml schema(xsd file) to java classes |  |
| schemaGen-  to generate xml schema/XSD file from java classes | these xjc, schemagen tools are used for compilation purpose- like if we have xsd file , if we want to generate stubs  note, we can generate xml only if we annotate the java classes with jaxb annotations |
| Runtime api –  marshalling – means converting java objects to xml data & vice versa is called unmarshalling |  |

**Changes in Java SE:**

* In older versions of the JDK, JAXB tools like xjc and schemagen were included as part of the JDK.
* However, starting with Java SE 11, JAXB was removed from the standard JDK distribution.
* This was part of a broader effort to modularize the Java platform.

**How to Use xjc and schemagen Today:**

* To use xjc and schemagen, you'll need to add the appropriate JAXB dependencies to your project.
* For Maven, you would add dependencies related to jaxb-xjc to your pom.xml file.
* The xjc tool is contained within the jaxb-xjc.jar.
* Therefore, to run xjc from the command line, you will need to have the jaxb-xjc.jar file, and execute it using the java -jar command.
* Many build tools, have plugins that will handle the execution of the xjc tool, and schemagen tool.

<https://github.com/highsource/jaxb-tools>

Plugin to generate stubs from xsd file

|  |  |
| --- | --- |
| here xjb means xml to java binding, when xjc compiler reads xml file and creates java classes it will read this xjb file before creating these java classes |  |
| <plugin>                  <groupId>org.jvnet.jaxb2.maven2</groupId>                  <artifactId>maven-jaxb2-plugin</artifactId>                  <version>${maven-jaxb2-plugin.version}</version>                  <executions>                      <execution>                          <id>example-xml</id>                          <goals>                              <goal>generate</goal>                          </goals>                          <configuration>                              <generateDirectory>target/generated-sources/xjc/</generateDirectory>                              <schemaDirectory>${basedir}/src/main/resources/schemas/xsd</schemaDirectory>                              <schemaIncludes>                                  <include>\*\*/\*.xsd</include>                              </schemaIncludes>                              <bindingDirectory>${basedir}/src/main/resources/schemas/xjb</bindingDirectory>                              <bindingIncludes>                                  <include>\*\*/\*.xjb</include>                              </bindingIncludes>                          </configuration>                      </execution>                  </executions>                  <dependencies>                      <dependency>                          <groupId>org.glassfish.jaxb</groupId>                          <artifactId>jaxb-runtime</artifactId>                          <version>2.3.3</version>                      </dependency>                  </dependencies>              </plugin> | sample xjb file  <jaxb:globalBindings>  <xjc:simple />  <xjc:serializable uid="-1" />  <jaxb:javaType name="java.util.Calendar" xmlType="xs:dateTime"  parseMethod="javax.xml.bind.DatatypeConverter.parseDateTime"  printMethod="javax.xml.bind.DatatypeConverter.printDateTime" />  </jaxb:globalBindings>  </jaxb:bindings> |
| <dependency>  <groupId>javax.xml.bind</groupId>  <artifactId>jaxb-api</artifactId>  <version>2.3.1</version>  </dependency>  <dependency>  <groupId>org.glassfish.jaxb</groupId>  <artifactId>jaxb-runtime</artifactId>  <version>2.3.1</version>  </dependency> | **<java.version>11</java.version>**  **<jaxb.version>2.3.0</jaxb.version>**  <dependencies>  <dependency>  <groupId>javax.xml.bind</groupId>  <artifactId>jaxb-api</artifactId>  <version>${jaxb.version}</version>  </dependency>  <dependency>  <groupId>com.sun.xml.bind</groupId>  <artifactId>jaxb-core</artifactId>  <version>${jaxb.version}</version>  </dependency>  <dependency>  <groupId>com.sun.xml.bind</groupId>  <artifactId>jaxb-impl</artifactId>  <version>${jaxb.version}</version>  </dependency>  </dependencies> |

Sample marshalling & unmarshalling program

* 1. The JAXBContext class provides the client's entry point to the JAXB API. It provides an abstraction for managing the XML/Java binding information necessary to implement the JAXB binding framework operations: unmarshal, marshal and validate.
  2. A client application normally obtains new instances of this class using one of these two styles for newInstance methods, although there are other specialized forms of the method available:

JAXBContext.newInstance( "com.acme.foo:com.acme.bar" ) or JAXBContext.newInstance( com.acme.foo.Foo.class )

|  |  |
| --- | --- |
| @XmlRootElement(name = "student")  these elements are mandatory for marshalling |  |
| @XmlRootElement(name = "employee1")  // only with this anno this ele will become as a root ele in xml tag public class Employee {  @XmlElement(name = "employeeName") //if u don’t keep this anno this field will not be included in final xml  String **empName**;  @XmlElement(name = "empRollNumber")  Integer **rollNum**;  public Employee(String *empName*, Integer *rollNum*) {  this.**empName** = *empName*;  this.**rollNum** = *rollNum*;  }  public Employee() {  } }  result  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <employee1>  <employeeName>mani</employeeName>  <empRollNumber>20</empRollNumber>  </employee1> | sample marshalling code  @Component public class MyAutoRunner implements *CommandLineRunner* {  @Override  public void run(String... *args*) throws Exception {  System.***out***.println("fired this runner");   Employee e1=new Employee("mani",20);   *JAXBContext* jaxbContext = *JAXBContext*.*newInstance*(Employee.class);  *Marshaller* marshaller = jaxbContext.createMarshaller();  marshaller.setProperty(*Marshaller*.***JAXB\_FORMATTED\_OUTPUT***,true);  StringWriter writer = new StringWriter();  marshaller.marshal(e1,new StreamResult(writer));  System.***out***.println(writer.toString());   } }  hence these annotations are very much helpful while marshalling and un-marshalling the java objects |
| sample unmarshalling code  public static void unmarshal2(String *xmlPayload*) throws XMLStreamException,JAXBException {  *XMLInputFactory* xif=*XMLInputFactory*.*newFactory*();  xif.setProperty(*XMLInputFactory*.***SUPPORT\_DTD***,false);  xif.setProperty(*XMLInputFactory*.***IS\_SUPPORTING\_EXTERNAL\_ENTITIES***,false);   *XMLStreamReader* xmlStreamReader = xif.createXMLStreamReader(new StreamSource(new StringReader(*xmlPayload*)));  *JAXBContext* jaxbContext = *JAXBContext*.*newInstance*(Employee.class);  *Unmarshaller* unmarshaller = jaxbContext.createUnmarshaller();  JAXBElement<Employee> unmarshal = unmarshaller.unmarshal(xmlStreamReader, Employee.class);  System.***out***.println("unm using streamReader"+unmarshal.getValue()); } | |

jaxb annotations

This @XmlType annotation is used to **Control the order of xml elements in the marshalled result**

|  |  |
| --- | --- |
| When u use this anno- then the class will have its own complex type in generated WSDL and eventually in the SOAP message  @XmlRootElement(name = "book") – only with these jaxb ano objects can be serialized and marshalled vice versa | sample WSDL file  <xs:complexType name="employee">  <xs:sequence>  <xs:element minOccurs="0" name="employeeName" type="xs:string"/>  <xs:element minOccurs="0" name="empRollNumber" type="xs:int"/>  </xs:sequence>  </xs:complexType> |
| @XmlRootElement(name = "book") // Still want the root element to be <book>  @XmlType(name = "BookDetails", propOrder = {"title", "author", "isbn"})  public class Book {  private String title;  private String author;  private String isbn;  }  result  <book>  <title>The Hitchhiker's Guide to the Galaxy</title>  <author>Douglas Adams</author>  <isbn>978-0345391803</isbn>  </book>  notice:- the marshalled result is in the same order of what we gave in @XmlType | This @XmlType annotation can be used with the following annotations: [XmlRootElement](https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/XmlRootElement.html), [XmlAccessorOrder](https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/XmlAccessorOrder.html), [XmlAccessorType](https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/XmlAccessorType.html), [XmlEnum](https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/XmlEnum.html). However, [XmlAccessorOrder](https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/XmlAccessorOrder.html) and [XmlAccessorType](https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/XmlAccessorType.html) are ignored when this annotation is used on an enum type.  **Example 1:**Map a class to a complex type with xs:sequence with a customized ordering of JavaBean properties.  @XmlType(propOrder={"street", "city" , "state", "zip", "name" })  The @XmlType annotation in JAXB allows you to:   * **Control the order of elements** in the XML representation using the propOrder attribute. (ex:- which field should come 1st In the generated xml payload) * **Rename the XML type** associated with your Java class using the name attribute.(but @XmlRootElement will have higher priority)   important note  , we also used @XmlRootElement(name = "book").hence this was given the priority over @XmlType(name = "BookDetails")  @XmlType controls the name of the **XML schema type** that the class maps to, which is used when the class is referenced as a type within other complex types or in the generated schema.(I also didn’t understand) |
| @XmlType(propOrder = { "city" , "state", "zip", "name","street"}) @XmlRootElement(name="USAAddressoh") //- this anno is mandatory @XmlAccessorType(*XmlAccessType*.***FIELD***) public class USAAddress {   @XmlElement(name = "person name") – to rename xml field In marshalled req  String **name**;  String **city**;  String **state**;  String **street**;  Integer **zip**; | @XmlElement(name = "person name") – this anno is to rename the field in marshalled xml payload & can be kept on either field or on getter  @XmlAccessorType(*XmlAccessType*.***FIELD***) - this must be kept on class to indicate wheter @XmlElement anno kept on fields or are they kept on getters  Imp note  @XmlAccessorType(*XmlAccessType*.***FIELD***), @XmlElement(name = "person name") – **these 2 are optional /not mandatory , these 2 are to just to rename the fields in payload**  but when @XmlElement(name = "person name") is there then @XmlAccessorType is mandatory |

|  |  |  |
| --- | --- | --- |
| @XmlType(propOrder = {"creditCardInfo","amount"}) @XmlAccessorType(*XmlAccessType*.***FIELD***) @XmlRootElement(name = "paymentProcessorReq") public class PaymentProcessorRequest {   @XmlElement(name = "creditCardInfo")  private CreditCardInfo **creditCardInfo**;  @XmlElement(name = "amount")  private Double **amount**; | <paymentProcessorReq>  <creditCardInfo>  <address>Kavali</address>  <cardNumber>122112</cardNumber>  <expirtyDate>2025-05-10T20:20:13.976+05:30</expirtyDate>  <firstName>Mani</firstName>  <lastName>voleti</lastName>  <secCode>C</secCode>  </creditCardInfo>  <amount>120.2</amount>  </paymentProcessorReq> | *JAXBContext* jaxbContext = *JAXBContext*.*newInstance*(PaymentProcessorRequest.class); *Marshaller* marshaller = jaxbContext.createMarshaller(); marshaller.setProperty(*Marshaller*.***JAXB\_FORMATTED\_OUTPUT***,true); StringWriter writer = new StringWriter(); marshaller.marshal(paymentProcessorRequest,new StreamResult(writer)); System.***out***.println(writer.toString()); System.***out***.println("marshalled"); |

A class maps to either a XML Schema complex type or a XML Schema simple type. The XML Schema type is derived based on the mapping of JavaBean properties and fields contained within the class. The schema type to which the class is mapped can either be named or anonymous. A class can be mapped to an anonymous schema type by annotating the class with @XmlType(name="").

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

Annotations

|  |  |
| --- | --- |
| @WebService | @WebService(name = "MyService", targetNamespace = "http://example.com/services")  public class MyServiceImpl {  @WebMethod  public String sayHello(String name) {  return "Hello, " + name + "!";  }  It is used to define a Java class as a web service endpoint.  This tells the JAX-WS runtime that the class's methods should be exposed as web service operations  . The JAX-WS runtime will use this information to create a WSDL document that describes the web service, which can then be used by clients to interact with the service. |
| @WebMethod | This annotation marks the Java methods that should be exposed as web service operations. If @WebMethod is not used, then all public methods are exposed.  The JAX-WS runtime will use this information to create a WSDL document that describes the web service, which can then be used by clients to interact with the service. |
| @WebResult | method result |

|  |  |
| --- | --- |
| @WebParam | used to control the name of the web services request parameters in the wsdl/soap request? |
| @WebFault | |  |  | | --- | --- | | @WebFault(name = "MyCustomFault",  targetNamespace = "http://example.com/faults",  faultBean = "com.example.MyFaultBean")  public class MyCustomException extends Exception {}  public class MyFaultBean {  private String errorCode;  private String detailedMessage;  //getters setters | @WebMethod  public String performOperation(String input) throws MyCustomException {  if ("error".equals(input)) {  com.example.MyFaultBean faultBean = new com.example.MyFaultBean();  faultBean.setErrorCode("123");  faultBean.setDetailedMessage("Invalid input provided.");  throw new MyCustomException("Operation failed.", faultBean);  }  return "Operation successful: " + input;  } |   name = "MyCustomFault": Specifies the name of the fault in the WSDL.  targetNamespace = "http://example.com/faults": Defines the XML namespace for the fault.  faultBean = "com.example.MyFaultBean": Indicates that MyFaultBean will provide the detailed fault information.  **MyFaultBean:**   * This is a simple Java bean that holds the error details (errorCode and detailedMessage) * it provides getter and setter methods for the fault information.   what happens if we don’t annotate the exception class with @WebFault  **Default Exception Handling:**   * Without @WebFault, JAX-WS will resort to its default exception handling mechanism. * This means that the MyCustomException will still be propagated to the client, but it will be represented as a generic SOAP fault. * The specific details of the exception, such as the errorCode and detailedMessage from the MyFaultBean, will likely not be included in a structured, easily accessible way within the SOAP fault. * The WSDL (Web Services Description Language) will also not contain the custom fault definition that @WebFault would have generated.   **Loss of Custom Fault Information:**   * The client will receive a less informative SOAP fault, making it more difficult to understand the cause of the error. * The client might only receive a generic fault message, rather than the specific error codes and detailed messages you intended. * The client will not have a way to know the structure of the MyFaultBean as it will not be defined in the WSDL.   ex:- if we annotate class with @WebFault, then only this MyFault class will present in WSDL along with all the fields ,  then when producer throws exception, client will have idea of those 2 params and client can fetch values from those parameters,  If u don’t declare with any anno , then this class will not at all be included in WSDL, when exception thrown, client doesn’t know about the structure of this class and he cant fetch the data from those 2 variables, hence always better to annotate the class with @WebFault , ensure this class info is present in WSDL |
| @SOAPBinding | @WebService  @SOAPBinding(style = SOAPBinding.Style.DOCUMENT,  use = SOAPBinding.Use.LITERAL,  parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)  public class MyWebService {  this defines soap message format structure   style: Defines the encoding style of the SOAP message (e.g., RPC or DOCUMENT).   |  |  | | --- | --- | | RPC (remote procedure call- we should write the payload with functions ) | DOCUMENT | | **RPC (Remote Procedure Call):**   * **Concept:**   + The RPC style treats the web service as a set of remote procedures or functions.   + The SOAP message is structured to resemble a function call, with parameters and return values.   + It focuses on the operations (methods) of the web service. * **Characteristics:**   + Oriented towards operations: The SOAP message reflects the method calls.   + Tightly coupled: The WSDL (Web Services Description Language) tends to be tightly coupled to the method signatures.   + Older style: While still used, it's generally considered less flexible than the DOCUMENT style. * **Example:**   + A SOAP request might look like:   <soap:Body>  <addNumbers>  <a>10</a>  <b>20</b>  </addNumbers>  </soap:Body>   * + This clearly resembles a call to an "addNumbers" function. |  **Concept:**   * The DOCUMENT style treats the web service as a provider of XML documents. * The SOAP message's body contains an XML document that represents the data being exchanged. * It focuses on the data being exchanged, rather than the operations.    **Characteristics:**   * Oriented towards data: The SOAP message contains XML documents. * Loosely coupled: The WSDL is more flexible and less tied to specific method signatures. * Modern style: Generally preferred for modern web services. * Strong schema usage: Typically uses XML schemas to define the structure of the XML documents.    **Example:**   * A SOAP request might look like:   <soap:Body>  <Order>  <OrderID>123</OrderID>  <CustomerName>John Doe</CustomerName>  </Order>  </soap:Body>   * This represents an "Order" document being sent to the web service. |    use: Specifies how the data is formatted within the SOAP message (e.g., LITERAL or ENCODED).   |  |  | | --- | --- | | LITERAL (here we didn’t mention the data types) | ENCODED (here while sending data we have to tell what is the data type of each data) | | <soap:Body>  <someInteger>123</someInteger>  <someString>Hello</someString>  </soap:Body> | <soap:Body>  <someInteger xsi:type="xsd:int">123</someInteger>  <someString xsi:type="xsd:string">Hello</someString>  </soap:Body> | |  "Literal" means that the SOAP message's data types adhere strictly to the XML Schema definitions provided in the WSDL (Web Services Description Language).   In simpler terms, the XML in the SOAP message directly reflects the XML Schema.   This approach is generally preferred because it promotes interoperability and allows for easy validation of SOAP messages against the schema.  Interoperable: Widely supported and recommended by standards like WS-I Basic Profile.  The values "123" and "Hello" are directly represented, and their types are defined by the associated XML Schema. |  **Concept:**   * "Encoded" means that the SOAP message's data types are serialized according to the SOAP 1.1 encoding rules. * This approach is older and less commonly used today. * It relies on rules within the SOAP 1.1 specification to define how data types are represented.    **Key Characteristics:**   * Relies on SOAP 1.1 encoding rules. * Less interoperable: Can lead to compatibility issues between different SOAP implementations. * Less common: Not recommended for new web services.   Notice the xsi:type attributes. These attributes explicitly specify the data types using the XML Schema instance namespace. This is characteristic of encoded SOAP messages. |    parameterStyle: Determines how method parameters are represented in the SOAP message (e.g., WRAPPED or BARE).   |  |  | | --- | --- | | wrapped (here all the method params are inside a method name /operation name tag) | bare | | <soap:Body>  <addNumbers>  <a>10</a>  <b>20</b>  </addNumbers>  </soap:Body> | <soap:Body>  <a>10</a>  <b>20</b>  </soap:Body> | |  In the "wrapped" style, all method parameters are enclosed within a single, named element. This element typically corresponds to the operation's name.   This style is very common, and often the default, because it helps to create more structured and organized SOAP messages.   It tends to produce cleaner and more interoperable WSDL (Web Services Description Language) documents.   **Example:**   * Imagine a web service operation called addNumbers(int a, int b). With "wrapped" style, the SOAP request body would look something like this: |  In the "bare" style, each method parameter becomes a direct child element within the SOAP message's body.   This style is less common and has stricter limitations. Notably, it's generally best suited for operations with a single parameter.   It can produce WSDL documents that are less structured.   **Example:**   * Using the same addNumbers example, with "bare" style, the SOAP request body might look like this: |   To define the structure of your SOAP messages, ensuring compatibility with different clients and systems. |

Logging annotations

|  |  |
| --- | --- |
| @Features | this anno is used to log entire request and response in the log file  @Features(feature = “org.apache.cxf.feature.LoggingFeature”)  there are many features, one of the feature is logging feature  @WebService @Features(features = "org.apache.cxf.feature.LoggingFeature") public class StudentSOAPController {  //since we added anno on top of this class alone, now only that class method request and responses will be logged into console not others..  <http://localhost:8080/hello>  sample log from console  2025-04-01 12:59:19.598 INFO 13064 --- [nio-8080-exec-7] o.a.c.s.S.S.StudentSOAPController : Inbound Message  ----------------------------  ID: 10  Address: http://localhost:8080/hello  Encoding: UTF-8  Http-Method: POST  Content-Type: text/xml;charset=UTF-8  Headers: {accept-encoding=[gzip,deflate], connection=[Keep-Alive], Content-Length=[236], content-type=[text/xml;charset=UTF-8], host=[localhost:8080], SOAPAction=[""], user-agent=[Apache-HttpClient/4.5.5 (Java/17.0.12)]}  Payload: <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:con="http://controller.soapDemo.ampf.com/">  <soapenv:Header/>  <soapenv:Body>  <con:**getStudentName**/> //this is the method name  </soapenv:Body>  </soapenv:Envelope>  --------------------------------------  2025-04-01 12:59:19.613 INFO 13064 --- [nio-8080-exec-7] o.a.c.s.S.S.StudentSOAPController : Outbound Message  ---------------------------  ID: 10  Response-Code: 200  Encoding: UTF-8  Content-Type: text/xml  Headers: {}  Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Body><ns2:getStudentNameResponse xmlns:ns2="http://controller.soapDemo.ampf.com/"><return>**Millet Follower Mani**</return></ns2:getStudentNameResponse></soap:Body></soap:Envelope>  -------------------------------------- |

Maven plugins

|  |  |  |
| --- | --- | --- |
| to gen the stubs from wsdl file  WSDL file will be given by soap producer | to gen the stubs from xsd file | to gen the stubs from json schema file |
| *use “****cxf-codegen-plugin****” plugin*  after taking those stubs , we should call those stubs  java classes and methods, then those real web service will be called | <plugin>  <groupId>org.jvnet.jaxb2.maven2</groupId>  <artifactId>maven-jaxb2-plugin</artifactId> | <plugin>  <groupId>org.jsonschema2pojo</groupId>  <artifactId>jsonschema2pojo-maven-plugin</artifactId> |

Why only CXF is famous

It goes beyond basic JAX-WS by supporting a wide array of WS-\* specifications like WS-Security, WS-ReliableMessaging, and WS-Policy

Bec CXF Is providing enterprise-grade web services requiring advanced features like security, reliable message delivery, and policy enforcement.

**Strong Integration Capabilities:**

* Spring Framework**:** CXF has excellent integration with the Spring Framework, which is widely used in Java enterprise applications. This simplifies configuration and management of web services within Spring-based applications.
* **OSGi:** CXF offers good support for OSGi, a modularity framework for Java, making it suitable for building modular and dynamic service-oriented architectures.

**4. Ease of Use and Development:**

* CXF provides simple APIs for quickly building code-first web services.
* It includes Maven plugins that simplify tooling integration, making the development process more efficient.
* Compared to the JAX-WS Reference Implementation (RI), CXF offers extensions that make it easier to use. For instance, it can automatically generate code for request and response bean classes, and it doesn't always require a WSDL for simple cases.

**Flexible Architecture:**

* CXF has a modular and pluggable architecture, allowing developers to customize and extend the framework based on their specific requirements. This makes it adaptable to various use cases and integration scenarios.
* We can customize using interceptors, handler

**Performance:**

* Apache CXF is known for its performance capabilities and is built to scale, making it suitable for handling large volumes of requests in complex SOA implementations

**Versatility:**

* CXF is the only framework providing implementation for both jax-rs, jax-ws & u can use this CXF for both create web services and to consumer web services
* CXF can produce both JAX-RS and JAX-WS implementations from the same data model and service interface. This is a significant advantage, allowing developers to expose the same service using different protocols (SOAP and REST) based on the client's needs.
* It supports various protocols beyond SOAP and HTTP, including JMS and CORBA, providing flexibility for integration with diverse systems

This CXF will serialize and de-serialize the xml payload to java objects &

This CXF has a soap Engine – which is responsible to dispatch the incoming request to a specifc WS endpoint method

CXF provides 2 tools named

Java2wsdl – from java classes it will generate wsdl file as part of code first approach

Wsdl2Java – from wsdl file we are generating java classes

SOAP Producer app

Maven jars

We have a starter for both soap and rest

<https://cxf.apache.org/docs/springboot.html> read this for all spring related configuration

|  |  |
| --- | --- |
| main spring jar which will pull all the dep jars like (jakarta.xml.ws-api)  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web-services</artifactId> </dependency> | for WS security add this jar  <dependency>  <groupId>org.apache.cxf</groupId>  <artifactId>cxf-rt-ws-security</artifactId>  <version>3.2.1</version> </dependency> |
| <dependency>      <groupId>org.apache.cxf</groupId>      <artifactId>cxf-spring-boot-starter-jaxws</artifactId>      <version>3.1.12</version>  </dependency> | <dependency>      <groupId>org.apache.cxf</groupId>      <artifactId>cxf-spring-boot-starter-jaxrs</artifactId>      <version>3.1.12</version>  </dependency> |
| for spring boot 2.4.0 compatibility version is apache cxf 3.4.0  Since below starters are created by cxf team , we can fint that compatibility info in spring boot dependencies website <https://docs.spring.io/spring-boot/appendix/dependency-versions/coordinates.html>  so to find which CXF is the correct compatible version for spring boot 3.4 see both the jars released dates  for spring boot 3.4.4 use cxf 3.6.6 | |
| for spr boot 3.2.2 cxf is 4.0.0 | |

|  |  |
| --- | --- |
| <groupId>jakarta.xml.ws</groupId> <artifactId>jakarta.xml.ws-api</artifactId>  this is the major jar for anno like @Webservice & @WebMethod  this is the transitive jar from (spr-boot-starter-jaxws) |  |

|  |  |
| --- | --- |
| no need to add below jars, just stating it as a reference as main jars (EndpointImpl class is coming from this jar only)  bus class is from (cxf-core)  <dependency>  <groupId>org.apache.cxf</groupId>  <artifactId>cxf-rt-frontend-jaxws</artifactId>  <version>${project.version}</version> </dependency> | Here main interface is given by (jakarta.xml.ws-api) and implementation class by cxf  here ws-api is the core jar – in core jar only interfaces will be present like Endpoint (I), Provider(interface) – like JPA spec  cxf-rt-frontend-jaxws – is the implementation jar – like hibernate |
| java8 – java 17 changes  Earlier all the major annotations like @WebService(import javax.jws.WebService;) are from main javas rt.jar  but after java 8 many changes are there hence all these jars were removed from java, & we have to manually add those required jars |  |

Main property

When "**cxf.jaxrs.component-scan=true**" then it will scan the base packages and spring will discover those endpoints and it will publish (no need of cxf-servlet.xml like in spring)

property to create a JAX-RS endpoint from the auto-discovered JAX-RS root resources and providers which are marked as Spring Components (annotated with Spring @Component or created and returned from @Bean methods).

Use "**cxf.path**" property to customize a CXFServlet URL pattern.

Use "**cxf.servlet.init**" map property to customize CXFServlet properties such as "services-list-path" (available by default at  "/services"), etc.

Use "**cxf.servlet.loadOnStartup**" set loadOnStartup priority of the CXFServlet (by default, -1)

Use "**cxf.servlet.enabled**" enable/disable CXFServlet regsitration (since **3.3.12**/**3.4.5**[)](https://issues.apache.org/jira/issues/?jql=project+%3D+CXF+AND+fixVersion+%3D+3.5.0)

Use "**cxf.jaxrs.server.path**" property to customize a JAX-RS server endpoint address (default is "/").

JAX-RS root resources and providers annotated with JAX-RS @Path and @Provider and native CXF Providers annotated with CXF [@Provider](https://github.com/apache/cxf/blob/cxf-3.1.6/core/src/main/java/org/apache/cxf/annotations/Provider.java) can be auto-discovered.

Use "**cxf.jaxrs.component-scan**-**packages**" property to restrict which of the auto-discovered Spring components are accepted as JAX-RS resource or provider classes. It sets a comma-separated list of the packages that a given bean instance's class must be in. Note, this property, if set, is only effective if a given bean is a singleton. It can be used alongside or as an alternative to the "**cxf.jaxrs.component-scan**-**beans**" property. This property is available starting from CXF 3.1.11.

Use "**cxf.jaxrs.component-scan**-**beans**" property to restrict which of the auto-discovered Spring components are accepted as JAX-RS resource or provider classes. It sets a comma-separated list of the accepted bean names - the auto-discovered component will only be accepted if its bean name is in this list. It can be used alongside or as an alternative to the "**cxf.jaxrs.component-scan**-**packages**" property. This property is available starting from CXF 3.1.11.

Use "**cxf.jaxrs.classes-scan**" property to create a JAX-RS endpoint from the auto-discovered JAX-RS root resources and provider classes. Such classes do not have to be annotated with Spring @Component. This property needs to be accompanied by a "**cxf.jaxrs.classes-scan-packages**" property which sets a comma-separated list of the packages to scan.

Note that while "**cxf.jaxrs.component-scan**" and "**cxf.jaxrs.classes-scan**" are mutually exclusive, "**cxf.jaxrs.component-scan**" can be used alongside the "**cxf.jaxrs.classes-scan-packages**" property to enable the auto-discovery of the JAX-RS resources and providers which may or may not be marked as Spring Components.

If needed, instead of having the resources auto-discovered,  one can use Spring ImportResource annotation to import the existing JAX-RS contexts available on the classpath.

To change the context root for our app “**server.servlet.context-path=/hellows**”

Creating SOAP WS

|  |  |
| --- | --- |
| create a soap controller – unlike REST controller ( where we will define path then and there itself on top of controller class) for soap classes we will define the paths in endpoint beans  here we are just writing controller , not path   * + Since cxf works well with spring, we can annotate the every web service class with spring @Component annotations (we can make them as spring beans) hence added @PostConstruct method   + To log both payload and response, we need to add @Features annotation   import org.apache.cxf.feature.Features; import org.springframework.stereotype.Component;  import javax.annotation.PostConstruct; import javax.jws.WebMethod; import javax.jws.WebService; import java.util.Collections; import java.util.*List*;  @WebService @Component //now this is a spring bean because of stereotype annotation @Features(features = "org.apache.cxf.feature.LoggingFeature") public class EmployeeController {  *List*<Employee> **employeeList**;  @WebMethod  public String empDetails(String *empName*){  return "Best Employee name is santhoshi and " + *empName*;  }   @WebMethod  public *List*<Employee> getAllEmployees(){  return **employeeList**;  }   @PostConstruct  public void init(){  System.***out***.println("firing init method");  **employeeList**=*List*.*of*(  new Employee("mohan",200),  new Employee("mani",201));  } }  *//import jakarta.jws.WebMethod; //import jakarta.jws.WebService;* import javax.jws.WebMethod; import javax.jws.WebService;  @WebService  @Component  public class StudentSOAPController {  @WebMethod  public String getStudentName(){  return "Millet Follower Mani";  }  @WebMethod  public String getStudentAddress(){  return "charan from chirala";  } } | Here main interface is given by (jakarta.xml.ws-api) and implementation class by cxf  here ws-api is the core jar – in core jar only interfaces will be present like Endpoint (I), Provider(interface) – like JPA spec  cxf-rt-frontend-jaxws – is the impl jar – like hibernate which have ProviderImpl, EndpointImpl  Endpoint is the class which have all the info like url, controller obj & interceptor objects  javax.xml.ws.Endpoint (ws-api )  |  |  |  org.apache.cxf.jaxws.EndpointImpl |
| import javax.xml.ws.Endpoint; this is correct as per barath  import org.apache.cxf.Bus;  import org.apache.cxf.jaxws.EndpointImpl;  here all the cxf runtime environment will be in bus   * Publish the endpoint and attach with bus   import com.ampf.SOAP\_1\_JAXB\_SprDemo.controller.EmployeeController; import com.ampf.SOAP\_1\_JAXB\_SprDemo.controller.StudentSOAPController; import org.apache.cxf.*Bus*; import org.apache.cxf.jaxws.EndpointImpl; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration;  import javax.xml.ws.*Endpoint*;  @Configuration public class SOAPConfig {  @Autowired  *Bus* **bus**;  @Bean  public *Endpoint* studentEndPoint(){  EndpointImpl impl=new EndpointImpl(**bus**,new StudentSOAPController());  impl.publish("/hello");  return impl;  }   @Bean  public *Endpoint* empEndpoint(){  EndpointImpl impl= new EndpointImpl(**bus**,new EmployeeController());  impl.publish("/employee");  return impl;  } }  alternate ways – we don’t need to create beans – even if container executes above lines that is also ok  @Autowired *Bus* **bus**;  @Autowired //since both controllers are spring beans we can happily autowire them EmployeeController **employeeController**;  @Autowired StudentSOAPController **studentSOAPController**;  @PostConstruct public void init() throws IOException {  EndpointImpl impl=new EndpointImpl(**bus**,**studentSOAPController**);  impl.publish("/hello");  System.***out***.println("fired post construct method");  EndpointImpl impl2= new EndpointImpl(**bus**,**employeeController**);  impl2.publish("/employee"); } |

Where to find WSDL & sample program &how to test

WSDL– means Webservice description language (here webservice means a REST controller this file describes only about 1 REST controller

and all its methods and each method signature(parameter and its return types)

-separate WSDL file for each controller and this WSDL file will be auto generated u can just access in below path

If u 10 SOAP Controller’s then we will have 10 WSDL files- 1 WSDL file for each controller

& in soap ui if 1 soap controller is having 2 methods / 2urls = 2 separate windows in soap ui testing tool – bec each method will have a separate url & payload

|  |  |
| --- | --- |
| by default we can see all the and WSDL endpoint locations &  services/controllers in “/services” u can even access the wsdl file in this endpoint  every controller will have 1 WSDL file | whereas if we want to see these info in main url after port number  then add this prop “cxf.path=/” in ur spring properties file  then u can access all the WSDL files in localhost:8080/    to configure server context path add this property  “server.servlet.context-path=/hellows” |
| <wsdl:portType name="CustomerOrders">  <wsdl:operation name="NewOperation">  <wsdl:input message="tns:NewOperationRequest" />  <wsdl:output message="tns:NewOperationResponse" />  </wsdl:operation>  </wsdl:portType> | here operation means method in that class |
| <wsdl:service name="CustomerOrders">  <wsdl:port binding="tns:CustomerOrdersSOAP"  name="CustomerOrdersSOAP">  <soap:address location="http://www.example.org/" />  </wsdl:port>  </wsdl:service> | here service name is the controller name  here we have url stating at which address that soap app is invocable  means if u invoke this class and method it will internally hit the webservice present in that url |

How to test in soap ui

|  |  |
| --- | --- |
| We can test the soap endpoints only using SOAP UI because in soap UI we have the option of giving WSDL file location   * + While creating the soap ui project, we must import using WSDL file location   bec only WSDL file have info like method name, class name..     * + Only after importing you can fire the requests   In a soap controller we may have multiple methods, we should tell which method to fire,    here in this student wsdl file /controller , we have 2 soap methods, hence we are seeing 2 different payloads /requests (getStudentAddress(), getStudentName())  @WebService public class StudentSOAPController {  @WebMethod  public String getStudentName(){  return "Millet Follower Mani";  }  @WebMethod  public String getStudentAddress(){  return "charan from chirala";  } }    here we are calling another method called studentAddress  so for eath method separate payload, bec in each payload method name is different | select the method and fire u will get 200 as response |

Old xml declarations

<import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>

<context:component-scan base-package="com.example.service"/>

<jaxws:endpoint id="helloWorldService"

implementor="#helloWorldBean"

address="/HelloWorld"/>

<bean id="helloWorldBean" class="com.example.service.impl.HelloWorldServiceImpl"/>

<jaxrs:server id="userService" address="/users">

<jaxrs:serviceBeans>

<ref bean="userResource"/>

</jaxrs:serviceBeans>

</jaxrs:server>

<bean id="userResource" class="com.example.service.rest.UserResource"/>

</beans>

Security - Configure interceptors (security guards) for SOAP app

We have 2 types of interceptoprs – In & OUT interceptor

IN interceptor will intercept the incoming request and out …

Set both types of interceptors to Endpoint class (Endpoint is the class which have all the info like url, controller obj & interceptor objects)

Analogy:- in a temple, at every gate 1 security guard will be there he will check all people, lly

for every endpoint we can configure a interceptor (security guard) who can check for presence of credentials in header

User Name token profile authentication (old approach)

In this approach we will pass user name and password in header(worst approach because we are passing direct credentials in the soap header This below approach will force if they didn’t send then their req will be rejected

* 1. Add security dependency
  2. add the WSS4JInterceptors to the endpointImpl class (this IN interceptors(like sec guards) are responsible to validate the cred)

here each interceptor is to do a task – here this interceptor is for pass comparision

* 1. Create the password callback class and we should set the password to the password callback (and spring will internally compare this with received pass is same or not, so call back class res is just to set the pass, and these call back classes will fire for each and every)
  2. Run & test
* Here In inteceptor is the one which will check for headers in the incoming request

|  |  |
| --- | --- |
| add sec jars  <dependency>  <groupId>org.apache.cxf</groupId>  <artifactId>cxf-rt-ws-security</artifactId>  <version>3.2.1</version> </dependency> |  |
| after code changes while testing in soap ui, we have to change the soap header by entering the user name and new password  below is the old soap head without credentials | Below is the soap header with credentials |

|  |  |
| --- | --- |
|  |  |
|  |  |

below is new soap header we have kept both username and password in the header

|  |
| --- |
| <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:con="http://controller.ampf.com/">  <soapenv:Header>  <wsse:Security soapenv:mustUnderstand="1" xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">  <wsse:UsernameToken xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">  <wsse:Username>mohan</wsse:Username>  <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">radhika</wsse:Password>  </wsse:UsernameToken>  </wsse:Security>  </soapenv:Header>  <soapenv:Body>  <con:getStudentAddress/>  </soapenv:Body>  </soapenv:Envelope> |

|  |  |
| --- | --- |
| public void run(String... *args*) throws Exception {  System.***out***.println("bus got autowired with hashcode" + **bus**.hashCode());  EndpointImpl impl = new EndpointImpl(**bus**, new StudentSOAPController());  impl.publish("/student");  System.***out***.println("published StudentSOAPController at /student");   *//configuring sec using interceptors  Map*<String, Object> map = new HashMap<>();  map.put(ConfigurationConstants.***ACTION***, ConfigurationConstants.***USERNAME\_TOKEN***);  map.put(ConfigurationConstants.***PASSWORD\_TYPE***, WSConstants.***PW\_TEXT***);  map.put(ConfigurationConstants.***PW\_CALLBACK\_CLASS***, MyUTPasswordTokenCallback.class.getName());   WSS4JInInterceptor wss4JInInterceptor = new WSS4JInInterceptor(map);  impl.getInInterceptors().add(wss4JInInterceptor); } |  |
| public class MyUTPasswordTokenCallback implements *CallbackHandler* {   *Map*<String,String> **credentials** =new HashMap<>();  public MyUTPasswordTokenCallback(){  **credentials**.put("mani","santu");  **credentials**.put("mohan","radhika");   }  @Override  public void handle(*Callback*[] *callbacks*) throws IOException, UnsupportedCallbackException {  System.***out***.println("executing the callback class, num of callback classes are "+*callbacks*.**length**);  for (*Callback* callback: *callbacks*) {  WSPasswordCallback callback1 = (WSPasswordCallback) callback;  String userName = callback1.getIdentifier(); *//here this user name will be pulled from soap payload* String password = **credentials**.get(userName);  if (password!=null){  *//pass which we set will be compared with pass from header, if mismatch then exception*  callback1.setPassword(password);  System.***out***.println("setted the password to callback");  break;  }   }  } } | Here this callback class will be fired for every request  so this callback method will extract the username and pass from header and it will compare with acutal pass which we sets  we should set the correct password to callback  if it founds any diff between header and what we set then it will throw exception |

Which part of the soap message contains the User Name Token Profile XML elements?

SOAPHeader

Consuming SOAP webservices programatically

Plugin to Gen Stubs from WSDL file

We should generate the stubs then controller classes will be generated, these stubs are useful when we want to consume webservices these stubs are required- bec we will call the methods present in the classes generate by stubs

|  |  |
| --- | --- |
| maven plugin    sample generated stubs | here >${basedir} means directory where the pom.xml resides  <plugin>  <groupId>org.apache.cxf</groupId>  <artifactId>cxf-codegen-plugin</artifactId>  <version>3.3.0</version>  <executions>  <execution>  <id>generate-sources</id>  <phase>generate-sources</phase>  <configuration>  <**!-- the final generate java classes/stubs will be kept in below folder here proj.build.di means target folder, we have to manually copy the generated stubs to src/main/java note- don’t change the package name while pasting to avoid errors** -->  <sourceRoot>${project.build.directory}/generated/cxf</sourceRoot>  <wsdlOptions>  <wsdlOption>  <!--here we are telling to pickupt the WSDL file from this location -->  <wsdl>${basedir}/src/main/resources/CustomerOrders.wsdl</wsdl>  <wsdlLocation>classpath:CustomerOrders.wsdl</wsdlLocation>  </wsdlOption>  </wsdlOptions>  </configuration>  <goals>  <goal>wsdl2java</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins> |
|  |  |
|  |  |

Sample program to consume web service (below says from service we will get controller , from contrller we will get client)

**service 🡪 Controller 🡪 client 🡪 Endpoint (we should configure all the in/out interceptors to this endpoint only)**

public class EmpWebSvcConsumer {  
  
 public static void main(String[] *args*) throws MalformedURLException {  
 EmployeeControllerService svc= new EmployeeControllerService(new URL("http://localhost:8080/employee?wsdl")); //while consuming the service WSDL location must be live  
 *EmployeeController* employeeController = svc.getEmployeeControllerPort();  
 *List*<Employee> allEmployees = employeeController.getAllEmployees();  
 System.***out***.println("printing all objects received from controller");  
 allEmployees.forEach(*e*-> System.***out***.println(*e*));  
 }  
}

* 1. Once SOAP producer server is up , we can get the WSDL file from - <http://localhost:8080/services> ,
  2. Generate stubs for that WSDL file using maven plugin “**cxf-codegen-plugin**” – by default stubs will be placed in “target” folder, manually copy those stubs and place it in src/main/java folder

Once we generate stubs from wsdl file, it will generate stubs classes as- EmployeeControllerService (from this class only we can get controller bean) , EmployeeController (this is the actual controller)

|  |  |
| --- | --- |
| generated stub service class will look like below & it can have constructors which can take URI  @WebServiceClient(name = "EmployeeControllerService",  wsdlLocation = "classpath:employee.wsdl",  targetNamespace = "http://controller.soapDemo.ampf.com/") public class EmployeeControllerService extends Service {  public EmployeeControllerService(URL wsdlLocation) {  super(wsdlLocation, SERVICE);  }  @WebEndpoint(name = "EmployeeControllerPort")  public EmployeeController getEmployeeControllerPort() {  return super.getPort(EmployeeControllerPort, EmployeeController.class);  }  } | * 1. Create \*Service(EmployeeControllerService) class object by passing Uri of webservice to EmployeeControllerService constructor   EmployeeControllerService svc=  new EmployeeControllerService(new URL("http://localhost:8080/employee?wsdl"));   * 1. Get the controller class object from service class object & then call those desired methods   *EmployeeController* employeeController = svc.getEmployeeControllerPort();  *List*<Employee> allEmployees = employeeController.getAllEmployees();  So we should get the controller class object using this service class , then only we can hit those methods |

Common error scenarios

* 1. Changing the package of stubs-

Source code is in diff package – com.tcs & we have generated those stubs and while moving – if we move those stubs to “com.infosys”

And while making service call- we are calling using url and while binding data to our stubs it will fail

So make sure source code of current running server and our stubs should be in same package

Consuming with security

Analogy:- in a temple, at every gate 1 security guard will be there he will check all people,

lly for every endpoint we can configure a interceptor (security guard) – different interceptor for diff endpoint

While consuming a web service , we should set the user name and password prop to a out interceptor (and those prop will be sent as part of header)

1. User name we will set normally, but
2. Password we will set in the handler class (instead of setting then and there it self, we are creating a callback class and fetching all usernames & setting the pass to particular user)
3. We are setting all these user name and “pass callback class” prop to an out interceptor

Now all these prop (user name and pass will be set to header ) and then the full payload will be sent

* Here out interceptor is the one which will add the credentials to the headers (while sending the req, it will add those headers to the payload)

Consuming without passing crede in header will throw exceptions as below

|  |  |
| --- | --- |
| StudentSOAPControllerService svc=new StudentSOAPControllerService(  new URL("http://localhost:8080/student?wsdl")); *StudentSOAPController* controller = svc.getStudentSOAPControllerPort(); String studentAddress = controller.getStudentAddress(); System.***out***.println("web service result is -->"+studentAddress); |  |
| Jun 02, 2025 4:06:27 PM org.apache.cxf.wsdl.service.factory.ReflectionServiceFactoryBean buildServiceFromWSDL  INFO: Creating Service {http://controller.ampf.com/}StudentSOAPControllerService from WSDL: http://localhost:8080/student?wsdl  Exception in thread "main" jakarta.xml.ws.soap.SOAPFaultException: A security error was encountered when verifying the message  at org.apache.cxf.jaxws.JaxWsClientProxy.mapException(JaxWsClientProxy.java:195)  at org.apache.cxf.jaxws.JaxWsClientProxy.invoke(JaxWsClientProxy.java:145)  at jdk.proxy2/jdk.proxy2.$Proxy33.getStudentAddress(Unknown Source)  at com.ampf.consumers.MySOAPConsumer.main(MySOAPConsumer.java:14)  Caused by: org.apache.cxf.binding.soap.SoapFault: A security error was encountered when verifying the message  at org.apache.cxf.binding.soap.interceptor.Soap11FaultInInterceptor.unmarshalFault(Soap11FaultInInterceptor.java:87) | |

Now we have to call a webservice provider with credentials in header

|  |  |
| --- | --- |
| public static void main(String[] *args*) throws MalformedURLException {   StudentSOAPControllerService svc=new StudentSOAPControllerService(new URL("http://localhost:8080/student?wsdl"));  *StudentSOAPController* controller = svc.getStudentSOAPControllerPort();   *Client* client = ClientProxy.*getClient*(controller);  *Endpoint* endpoint = client.getEndpoint();   *Map*<String,Object> props=new HashMap<String,Object>();  props.put(WSHandlerConstants.***ACTION***,WSHandlerConstants.***USERNAME\_TOKEN***);  props.put(WSHandlerConstants.***USER***,"mani");  props.put(WSHandlerConstants.***PASSWORD\_TYPE***, WSS4JConstants.***PW\_TEXT***);  props.put(WSHandlerConstants.***PW\_CALLBACK\_CLASS***,MyPasswordCallback.class.getName());   *//We must add a task to every interceptor/security guard* WSS4JOutInterceptor outInterceptor =new WSS4JOutInterceptor(props);  *//add the interceptor to endpoint* endpoint.getOutInterceptors().add(outInterceptor);    String studentAddress = controller.getStudentAddress();  System.***out***.println("web service result is -->"+studentAddress); } | here out interceptor will get the username and password from callback classes  and it will set to header |
| public class MyPasswordCallback implements *CallbackHandler* {  @Override  public void handle(*Callback*[] *callbacks*) throws IOException, UnsupportedCallbackException {  *//Retrieve the user name and set the password* System.***out***.println("Executing callback class to set pass");  for (*Callback* c:*callbacks*){  WSPasswordCallback myCallback = (WSPasswordCallback) c;  String identifier = myCallback.getIdentifier();  if(identifier.equalsIgnoreCase("mani")){  myCallback.setPassword("santu");  System.***out***.println("password have been set to the user name, now about to fire webservice");  break;  }  }  } } |  |

|  |
| --- |
| passed payload (note see the producer log as per log, producer got both username and pass from the header )  ID: 2  Address: http://localhost:8080/student  Encoding: UTF-8  Http-Method: POST  Content-Type: text/xml; charset=UTF-8  Headers: {Accept=[\*/\*], connection=[Upgrade, HTTP2-Settings], Content-Length=[720], content-type=[text/xml; charset=UTF-8], host=[localhost:8080], http2-settings=[AAEAAEAAAAIAAAAAAAMAAAAAAAQBAAAAAAUAAEAAAAYABgAA], SOAPAction=[""], upgrade=[h2c], user-agent=[Apache-CXF/4.0.4]}  Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Header><wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd" xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" soap:mustUnderstand="1">  <wsse:UsernameToken wsu:Id="UsernameToken-2d3cec01-95e9-4242-a9fb-65d335b012aa"><wsse:Username>mani</wsse:Username>  <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">santu</wsse:Password></wsse:UsernameToken></wsse:Security>  </soap:Header><soap:Body><ns2:getStudentAddress xmlns:ns2="http://controller.ampf.com/"/></soap:Body></soap:Envelope> |

Spring oxm – obj xml mapping

With Spring OXM module – we can marshal and unmarshal easily with 2 new interfaces – Marshaller, Unmarshaller

Spring is simplifying everything

|  |  |
| --- | --- |
| marshalling – means converting java objects to xml data & vice versa is called unmarshalling |  |
| import org.springframework.oxm.Marshaller;  import org.springframework.oxm.Unmarshaller;  import org.springframework.stereotype.Service;  import javax.xml.transform.stream.StreamResult;  import javax.xml.transform.stream.StreamSource;  import java.io.StringReader;  import java.io.StringWriter;  import java.io.IOException;  @Service  public class XmlService {  private Marshaller marshaller;  private Unmarshaller unmarshaller;  public void setMarshaller(Marshaller marshaller) {  this.marshaller = marshaller;  }  public void setUnmarshaller(Unmarshaller unmarshaller) {  this.unmarshaller = unmarshaller;  }  public String marshal(Person person) throws IOException {  StringWriter writer = new StringWriter();  marshaller.marshal(person, new StreamResult(writer));  return writer.toString();  }  public Person unmarshal(String xml) throws IOException {  StringReader reader = new StringReader(xml);  return (Person) unmarshaller.unmarshal(new StreamSource(reader));  }  } |  |

MTOM (Message Transmission and Optimization Mechanism.)

This MTOM is used to transfer the files

By default this MTOM feature is disabled

Practice this mtom

JAX-WS- Handlers

These handlers are used to address cross cutting concerns (this is similar to AOP)

This handler is also like a security guard, which will fire for every request and response

Similarities – both interceptors and handlers will be executed for incoming and outgoing requests, but still we will use handlers for cross cutting concerns

These SOAP handlers are standardized message interceptors

When to use- when u want to fetch something from header –like if u want cred from header or custom values for certain tags

|  |  |
| --- | --- |
|  | here this image showed 2 interceptors/Handlers (one configured at client another configured at server )  interceptor at client:- will be called 2 times – before sending the request to server and before accepting the response back from server    this image shows, this handler / interceptor will be exec for both req and resp |

Usages of JAX ws handlers are

* 1. For security purposes – Adding security tokens to headers (while sending response ) & checking for sec in header (while receving request) or performing authentication and authorization checks – u can write logic in soap handler to validate the header auth details, if failed, then we don’t even need to forward that req to our controller classes also
  2. Caching purpose- as every req goes through this handler, put a map in handler and cache all req- if same req comes again then load it from cache
  3. Logging - Logging incoming and outgoing SOAP messages.
  4. Custom error handling – when exception came during that endpoint execution, then handleFault() method will be automatically executed
  5. Performance optimization- caching the request & response based on the request parameter found in the header

There are 2 types of handlers,

1. soapHandler – these will get both soap headers, payload
2. logicalHandler – these will get only payload / soap body

Multiple handlers can be chained together to perform a sequence of operations on the message. The order of execution can be configured.

|  |  |
| --- | --- |
| soapHandler<SoapMessageContext>  this handler will be executed for both req, response | LogicalHandler<LogicalMessageContext> |
| methods()   * handleHeader() - this will be fired for both req and resp, this is related to mustunderstand attribute of any tag * boolean handleMessage() – this will **also be executed for both req and resp**   if this method returns false, then webservice endpoint will not be called at all   * handleFault() – this method will be called automatically when exception is thrown * close() – this is called only once | methods()   * handleMessage() * handleFault() * close() |
| These are protocol-specific and can access and modify the entire SOAP message, including headers and the body. | These are protocol-agnostic and operate only on the payload (the business data in the SOAP body), without direct access to protocol-specific parts like SOAP headers. |

If message\_outbound\_property= true – then it is a response message, false means it is request message

QQ- if soap handler can intecept and get both message and header then why we need LogicalHandler?

What is handleMessage() – this will get the full soap envolope – here envolope means both payload and headers

What is handlerHeader() method

|  |  |
| --- | --- |
| <soap:Header>   <m:Trans xmlns:m="https://www.w3schools.com/transaction/"   soap:mustUnderstand="1">234   </m:Trans> </soap:Header>  if client send this   soap:mustUnderstand="1"> attribute then server has to process that header | inside soap header  If any tag is having mustUnderstand attribute (that attribute can have values 0 or 1)  1 means this soap header should be processed by web service provider/project in which there is a soap provider    if 1 means- we should return that tag “Trans” ,then cxf will understand that soap provider processed this particular tag  1 means we have to process that header either in handler/ web service endpoint  else exception/ SOAP fault will be thrown |

|  |  |
| --- | --- |
| package com.ampf.handler;  import jakarta.xml.soap.*Node*; import jakarta.xml.soap.SOAPException; import jakarta.xml.soap.*SOAPHeader*; import jakarta.xml.soap.*SOAPMessage*; import jakarta.xml.ws.handler.*MessageContext*; import jakarta.xml.ws.handler.soap.*SOAPHandler*; import jakarta.xml.ws.handler.soap.*SOAPMessageContext*; import org.springframework.stereotype.Component;  import javax.wsdl.*Message*; import javax.xml.namespace.QName; import java.util.*Iterator*; import java.util.*Set*;  @Component public class IdCardCheckerSOAPHandler implements *SOAPHandler*<*SOAPMessageContext*> {   @Override  public *Set*<QName> getHeaders() {  return *Set*.*of*();  }   @Override  public boolean handleMessage(*SOAPMessageContext soapMessageContext*) {  boolean isResponse = (Boolean)*soapMessageContext*.get(*MessageContext*.***MESSAGE\_OUTBOUND\_PROPERTY***);  if(isResponse){  System.***out***.println("executing handler class- handleMessage method - for response ");  }else {  System.***out***.println("executing handler class- handleMessage method - for request");  *SOAPMessage* message = *soapMessageContext*.getMessage();  try{  *SOAPHeader* header = message.getSOAPPart().getEnvelope().getHeader();  *Iterator*<*Node*> childElements = header.getChildElements();  System.***out***.println("iterating all header");  while (childElements.hasNext()){  *Node* currentHeader = childElements.next();  System.***out***.println("header name -->"+currentHeader.getLocalName()+" header value -->"+currentHeader.getValue());  }  } catch (SOAPException *e*) {  throw new RuntimeException(*e*);  }   }  return true;  }   @Override  public boolean handleFault(*SOAPMessageContext soapMessageContext*) {  System.***out***.println("executing handler class- handleFault method");  return false;  }   @Override  public void close(*MessageContext messageContext*) {   } } |  |

Linking that interceptor to that endpoint

|  |  |
| --- | --- |
| @Component public class MySOAPEndpointRegistrator implements *CommandLineRunner* {  @Autowired  DoctorController **doctorController**;   @Autowired  ActorController **actorController**;   @Autowired  *Bus* **bus**;   @Autowired  IdCardCheckerSOAPHandler **mySOAPHandler**;   @Override  public void run(String... *args*) throws Exception {   EndpointImpl endpoint =new EndpointImpl(**bus**,**doctorController**);  endpoint.publish("/doctors");  System.***out***.println("registered doctor controller");  endpoint.getBinding().setHandlerChain(*List*.*of*(**mySOAPHandler**));  System.***out***.println("configured mySOAP Handler for endpoint1");    EndpointImpl endpoint1 =new EndpointImpl(**bus**,**actorController**);  endpoint1.publish("/actors");  endpoint1.getBinding().setHandlerChain(*List*.*of*(**mySOAPHandler**));  System.***out***.println("configured mySOAP Handler for endpoint2");  }  }  we should configure this handler to specific url basis ,  if we configure handler for /doctors , means it will fire only for /doctors, not for /actors  if we want this handler for multiple endpoints, then cfg same endpoint for all urls | so here when we pass the header , we can get the header values from handler class  console output  executing handler class- handleMessage method - for request  iterating all header  header name -->null header value -->    header name -->companyName header value -->Tcs  header name -->null header value -->companyName>  header name -->null header value -->    hitting controller method -get doctor names.....  executing handler class- handleMessage method - for response |

Read this <https://www.w3schools.com/xml/xml_soap.asp>

SOAP faults

|  |  |
| --- | --- |
| In real time, producer will not create and throw a soap fault,  instead java developers will throw a run time exception, SOAP api will catch that and it will convert that exception into soap fault,  and framework will send that soap fault to consumers   * so throw a runtime exception- it will be auto translated to soap fault   Even soap fault will also be present in soap body | @Component @WebService public class DoctorController {   @WebMethod  public *List*<String> getAllDoctorNames() {  int x = 0;  System.***out***.println("hitting controller method -get doctor names.....");  if (x == 0) {  throw new RuntimeException("I am closing hospital");  }  return *List*.*of*("T.NagaLakshmi", "Munny", "seema kumari");  } } |
| here fault code shows server, means server side exception | here fault code soap:client means client exception  XML data request is case sensitive , so even if I change a little meaning will change  here I changed the case to small “n” server considered it as new method and it thrown exception saying , some issue is there in client |
|  |  |

|  |  |
| --- | --- |
| <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">  <soapenv:Body>  <soapenv:Fault>  <faultcode>QName</faultcode>  <faultstring>String</faultstring>  <faultactor>URI</faultactor>  <detail> </detail>  </soapenv:Fault>  </soapenv:Body>  </soapenv:Envelope> |  **<faultcode> (Mandatory):** Provides an algorithmic error code that can be processed by software. It uses a Qualified Name (QName).   * **Common Predefined Values:**   + VersionMismatch: Invalid namespace for the SOAP Envelope.   + MustUnderstand: A header element with mustUnderstand="1" was not processed.   + Client: The client sent an invalid request (e.g., malformed message, invalid parameters). This usually indicates a client-side error that requires correction from the client.   + Server: The server encountered an error and could not process the request, even though the request itself was valid. This indicates a server-side problem.    **<faultstring> (Mandatory):** A human-readable explanation of the fault. This is intended for end-users or developers for debugging.   **<faultactor> (Optional):** A URI that identifies the SOAP node that generated the fault. This is useful in multi-hop SOAP message paths to pinpoint where the error occurred. If the ultimate receiver generates the fault, this element is not strictly required but can be included.   **<detail> (Optional):** Carries application-specific error information related to the <body> element. This is where you can put custom XML structures that provide more granular details about the error that are relevant to your specific application logic. |

|  |  |
| --- | --- |
| <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">  <soapenv:Body>  <soapenv:Fault>  <faultcode>soapenv:**Client**</faultcode>  <faultstring>Invalid input parameter: customerId cannot be negative.</faultstring>  <faultactor>http://example.com/OrderService</faultactor>  <detail>  <ns1:OrderServiceFault xmlns:ns1="http://example.com/faults">  <ns1:errorCode>1001</ns1:errorCode>  <ns1:errorMessage>Customer ID provided was -5, which is invalid.</ns1:errorMessage>  <ns1:timestamp>2025-06-15T19:09:12Z</ns1:timestamp>  </ns1:OrderServiceFault>  </detail>  </soapenv:Fault>  </soapenv:Body>  </soapenv:Envelope> | below is server fault  <faultcode>soapenv:**server**</faultcode>  below means issue in client side  due to malformed xml..  <faultcode>soapenv:**Client**</faultcode> |